VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

Andrej Vojtuš

Bakalářská práce

Vedoucí práce: Ing. Pavel Dohnálek, Ph.D.

Ostrava, 2021

Abstrakt

Táto bakalárska práca popisuje priebeh mojej individuálnej odbornej praxe v spoločnosti profiq s.r.o. Náplňou mojej práce počas tejto praxe bola tvorba a automatizovanie testov pre modernú webovú aplikáciu. V tejto práci sa preto dotýkam problematiky testovania softvéru a QA, opisujem využité technológie a môj postup pri práci s nimi a v závere načŕtam súhrn získaných znalostí, znalosti ktoré mi chýbali a zhodnotenie výsledkov tejto praxe.

Kľúčové slová

odborná prax, automatické testy, regresné testovanie, JavaScript, webdriver.io, CI, docker

Abstract

This bachelor thesis describes the course of my individual professional practice in the company profiq s.r.o. The scope of my work during this practice was the creation and automation of tests for a modern web application. In this work I therefore touch on the issue of software testing and QA, I describe the technologies used and my approach to working with them, and in the end I outline a summary of acquired knowledge, knowledge that I lacked and evaluation of the results of this practice.

Keywords

professional practise, automatic tests, regression testing, JavaScript, webdriver.io, CI, docker

Poďakovanie

Rád by som sa týmto poďakoval spoločnosti profiq s.r.o., jej CEO Gáborovi Puhallovi a všetkým členom Planesty tímu za umožnenie absolvovania tejto odbornej praxe, mnoho novonadobudnutých skúseností a nespočet úsmevných chvíľ. Taktiež by som sa chcel poďakovať vedúcemu mojej práce, pánovi Ing. Pavlovi Dohnálkovi, Ph.D. za jeho čas pri konzultáciach a pomoc počas písania tejto bakalárskej práce.

Obsah

Zoznam použitých skratiek a symbolov

CI	–	Continuous Integration
CD	–	Continuous Deployment
QA	–	Quality Assurance
QC	–	Quality Control
FE	–	Frontend
BE	–	Backend
GUI	–	Graphical User Interface
HTTP	–	Hypertext Transfer Protocol
HTML	–	Hypertext Markup Language
REST	–	Representational state transfer
BE	–	Backend
CSV	–	Comma-separated values
BEO	–	Banquet Event Order
API	–	Application Programming Interface
DOM	–	Document Object Model

Zoznam obrázkov

Zoznam výpisov zdrojového kódu

Kapitola 1

Úvod

V akademickom roku 2020/2021 som absolvoval odbornú prax vo firme profiq s.r.o., kde som pracoval na pozícii junior software engineer so zameraním na QA. Túto možnosť som si zvolil preto, aby som overil a aplikoval moje teoretické znalosti získané počas štúdia a taktiež mi táto praktická skúsenosť môže pomôcť v ďalšom zamestnaní po skončení štúdia.

Spomedzi všetkých dostupných ponúk ma pri vyberaní zamerania najviac zaujalo QA, a to kvôli tomu, že som v tomto odvetví som ešte nemal žiadne predchádzajúce skúsenosti a mala by to byť rozhodne jedna zo základných schopností každého dobrého vývojára, keďže vytvárať kvalitný a bezproblémový softvér nie je jednoduchý úkon.

Táto pozícia bola vytvorená pretože po zahájení spolupráce na vývoji aplikácie s novým klientom bolo potrebné zabezpečiť priebežnú kontrolu kvality tohto softvéru. Keďže pre tento projekt doposiaľ neboli navrhnuté žiadne špecifikácie pre tvorbu testovacích scenárov, mal som za úlohu ich vytvoriť a následne zautomatizovať aby nebolo potrebné manuálne overovať či softvér pracuje správne aj po každom nasadení zmien.

Na začiatku tejto bakalárskej práce predstavujem spoločnosť profiq s.r.o., jej históriu a zameranie. Taktiež som v úvodnej časti rozpracoval projekt, na ktorom som sa podieľal a popis náplne mojej práce. Nasledujúca kapitola obsahuje krátky výklad významu QA a softvérového testovania zo všeobecného hľadiska. V neposlednom rade opisujem samostatné úlohy riešené počas praxe spolu s technológiami, ktoré som pri ich riešení použil. Záver tejto práce sa napokon zaoberá mojimi dosiahnutými výsledkami a ich zhodnotením, znalosťami získanými v škole, ktoré som dokázal uplatniť v priebehu tejto praxe a taktiež znalosťami ktoré mi chýbali.

Kapitola 2

Popis odborného zamerania firmy a môjho pracovného zaradenia

Cieľom tejto kapitoly je nielen sprístupnenie stručných informácií o spoločnosti, v ktorej som bol zamestnaný a vykonával odbornú individuálnu prax, ale zameriava sa aj na obsahovú náplň mojej práce a popis projektu, pod ktorý som bol zaradený.

2.1 Spoločnosť profiq s.r.o.

Spoločnosť profiq s.r.o. bola založená v roku 2010 Gáborom Puhallom a Rastislavom Kanóczom so sídlom v Prahe. Pôvodným zameraním firmy bolo poskytovanie služieb v oblasti outsourcingu testovania softvéru, avšak neskôr začala poskytovať aj outsourcing full-stack softvérového vývoja. V súčastnosti spolupracuje aj s mnohými inovatívnymi zahraničnými spoločnosťami, primárne z USA (Silicon Valley, Colorado a Utah), ktorým poskytuje softvérové riešenia na mieru podľa ich požiadavkov. Medzi najznámejších klientov aktuálne patria Avast, Liferay, ForgeRock, DivvyPay, Sencha a mnoho iných. [1]

V dnešnej dobe spoločnosť profiq s.r.o. disponuje troma pobočkami v Českej Republike (Praha, Ostrava a Nový Jičín) a dvoma zahraničnými pobočkami, jednou v Martine na Slovensku a druhou v Madride vo Španielsku. V Ostravskej pobočke taktiež prevádzkuje program *Student Pool*, ktorý umožňuje študentom z VŠB-TUO získať nové praktické skúsenosti pomocou trojmesačnej stáže. [2]

2.2 Projekt Planesty

Americký startup[1], pre ktorý som pracoval, sa zaoberá vývojom aplikácie slúžiacej na jednoduchú správu a plánovanie konferencií, koncertov a iných udalostí. S firmou profiq začala táto americká

[1] začínajúca spoločnosť s inovatívnou podnikateľskou koncepciou, rýchlym vývojom a veľkým potenciálom na hospodársky rast

Obr. 2.1: Logo spoločnosti profiq s.r.o.

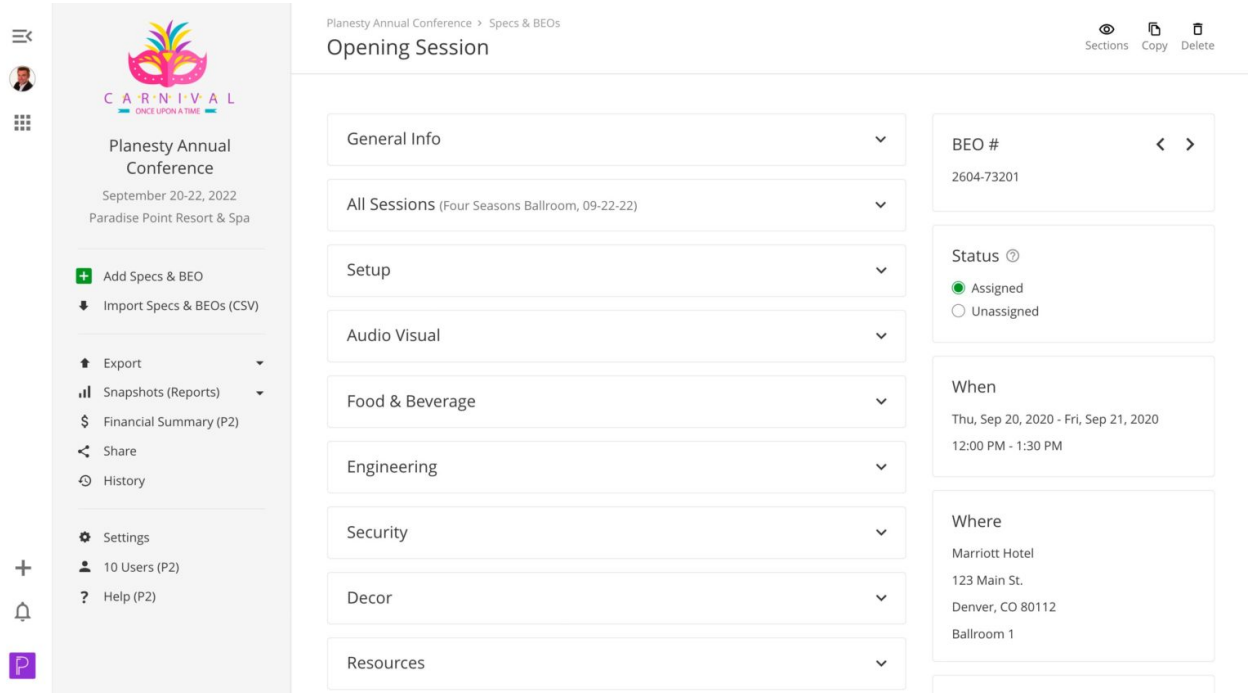
firma spolupracovať za účelom prepracovania predchádzajucej verzie bežiacej na PHP do modernejších technológií ako napríklad React, Django, Material UI a GraphQL.

Táto aplikácia je navrhnutá ako „all-in-one“ riešienie pre väčšinu problémov spojených s plánovaním udalostí, špecificky: [3]

- Tvorba BEO - dokumentu obsahujúceho všetky podrobnosti o udalosti ako sú
  - názov, miesto a čas konania
  - poradie a načasovanie udalostí
  - logistika parkovania a prepravy
  - výber a rozmiestnenie jedál a nápojov
  - zoznam a opis poskytovaných služieb
  - audiovizuálne vybavenie a jeho rozmiestnenie
  - kontaktné informácie dodávateľov
  - celkové náklady a ich rozpis
- Tvorba programu/harmonogramu
- Tvorba špecifikácii udalosti
- Spravovanie a automatizácia vytvárania zoznamov úloh
- Spravovanie rozpočtu
- Organizácia inventáru

- Spravovanie kontaktov klientov, dodávateľov a podobne
- Zdieľanie súborov a dokumentov

Aplikácia je rozdelená na takzvané workspaces, ktoré umožňujú jednej osobe pracovať na viacerých rozličných udalostiach naraz, zatiaľ čo všetky dáta a informácie ostávajú oddelené a prístupné iba osobám s potrebnými oprávneniami.



Obr. 2.2: Uživateľské rozhranie aplikácie Planesty

2.3 Moje pracovné zaradenie

Počas mojej bakalárskej praxe som bol v Ostravskej pobočke spoločnosti profiq s.r.o. súčasťou 6-členného developerského tímu na pozícii junior software engineer so zameraním na QA. Náplňou mojej práce bol hlavne návrh testovacích scenárov pomocou exploratory testingu, vývoj automatických testov pre frontend a backend webovej aplikácie a ich nasadenie do CI/CD procesu.

Kapitola 3

Význam QA a QC

Skôr ako sa dostaneme k problematike riašenia konkrétnych úloh je potrebné vysvetliť si niektoré pojmy, vďaka ktorým pochopíme význam mojej práce a celkovú funkciu QA v rámci vývoja softvéru.

QA, alebo *„quality assurance“* je súbor procesov zabezpečujúcich konzistentnú a vysokú kvalitu vyvíjaného softvéru a slúži ako prevencia problémov spôsobených vydaním chybnej aktualizácie. K dosiahnutiu efektivity QA je nevyhnutnou požiadavkou, aby ho praktikoval celý tím vývojárov. [4]

Ďalší pojem, s ktorým sa môžeme stretnúť v súvislosti s QA je QC, alebo *„quality control“*. Cieľom QC je pomocou testovania nájsť chyby a odchýlky od špecifikácie a zabezpečiť, aby sa znova neopakovali. Na rozdiel od QA je to reaktívny proces, ktorý konzumuje viac času a je závislý na tíme testerov.

3.1 Testovanie softvéru

Testovanie je neoddeliteľná súčasť vývoja softvéru. Čím je softvér komplexnejší, tým viac je pravdepodobné, že zdanlivo jednoduchá zmena kódu alebo oprava bugu sa môže v aplikácii negatívne prejaviť na inom nečakanom mieste. Dokážeme vďaka nemu predísť mnohým chybám, ktoré nás v budúcnosti môžu stáť veľa času a financií. Možno najznámejší príklad je takzvaný Y2K[1] bug.

Keďže v počiatkoch digitálnej éry mali počítače veľmi obmedzenú pamäť a výpočtovú silu, programátori často nachádzali kreatívne spôsoby ako programy optimalizovať. Jedným z týchto spôsobov bolo aj ukladanie roku do 2 číslic miesto 4. Rok 1977 bol teda uložený ako 77 a podobne. Toto však malo nečakané následky, pretože mnoho programov ostalo nezmenených až do prelomu milénia a nebolo jasné, čo sa stane ak začnú vykonávať kalkulácie s rokom 00 alebo 01. Dnes sa odhaduje, že na opravu tohto bugu bolo celosvetovo minutých niekoľko miliárd dolárov. Spravidla teda cena opravy bugu logaritmicky rastie čím neskôr ho opravíme, a preto by bolo ideálne sa im vyhnúť už pri špecifikácii, to však bohužiaľ zatiaľ nie je možné.

[1] Year 2000

V predošlom texte bol viackrát uvedený pojem bug, ktorý je potrebné pre lepšie pochopenie uvedenej problematiky objasniť. Ron Patton ho v knihe „Software Testing“ definoval nasledovne: [5]

1. *Softvér nerobí niečo, čo uvádza špecifikácia produktu.*
2. *Softvér robí niečo, čo uvádza špecifikácia produktu že by robiť nemal.*
3. *Softvér robí niečo, čo špecifikácia produktu neuvádza.*
4. *Softvér nerobí niečo, čo špecifikácia produktu neuvádza, ale mal by.*
5. *Softvér je ťažko pochopiteľný, ťažko použiteľný, pomalý alebo v očiach testera softvé bude koncový používateľ považovať tento produkt jednoducho za nesprávny.*

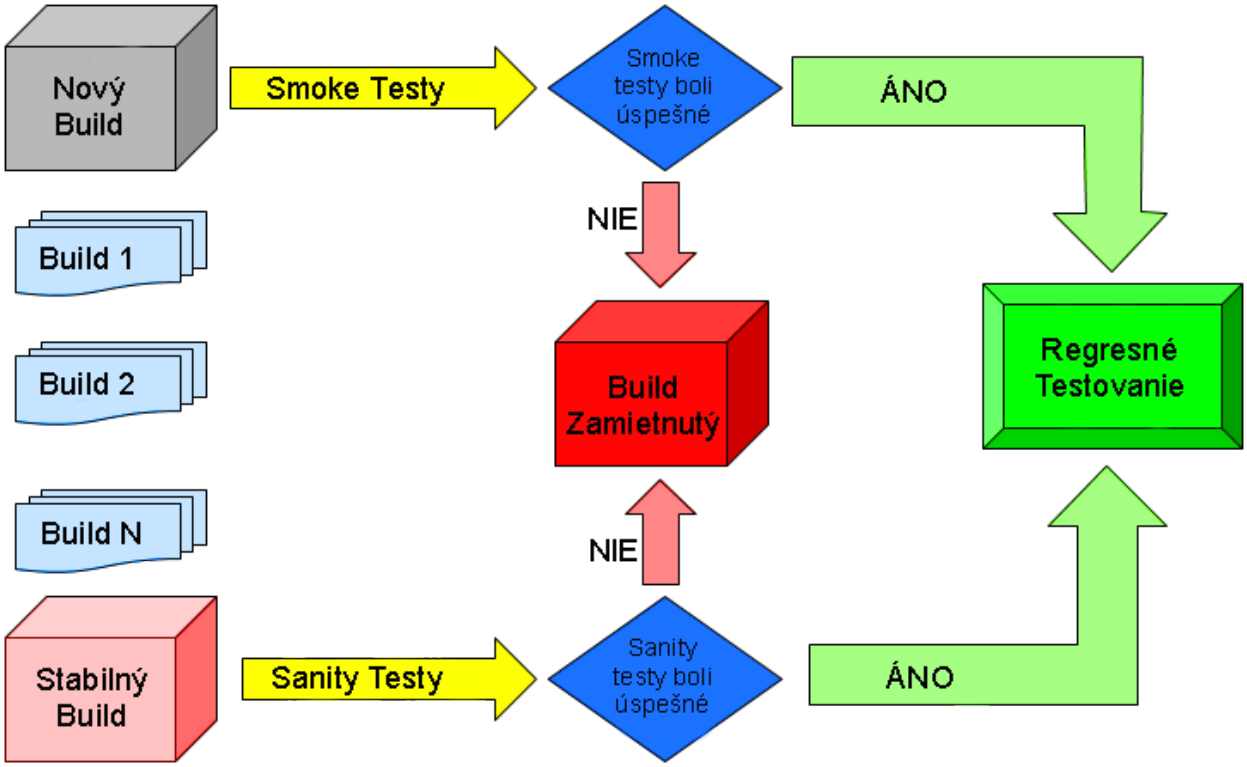
3.2 Typy testov

Aby sa ušetril čas testerov, jednotlivé testy sú rozdelené do troch typov podľa dôležitosti, menovite *„smoke testy“*, *„sanity testy“* a *„regression testy“*. Často býva dosť zložité ich rozlíšiť a preto je dôležité čo najlepšie popísať aký má toto rozdelenie význam.

Smoke testy – QA tím vykonáva smoke testy zakaždým, keď získa nový build softvéru. Jednoducho povedané, konajú sa vždy, keď vývojári zmenia kód a prevedú ho na aktualizovanú aplikáciu. Primárnym účelom smoke testov nie je vykonať dôkladné testovanie, ale zabezpečiť, aby základné funkcie fungovali bez problémov. Tento proces sa vykonáva pred vykonaním akýchkoľvek ďalších podrobných testov, vďaka čomu je možné odhaliť vážne chyby v najskorších fázach, odmietnuť build a odoslať kód na prepracovanie. Testeri tak neriskujú stratu času pri dlhom a komplikovanom testovaní softvéru, ktorý je pokazený.

Sanity testy – Cieľom sanity testov je skontrolovať, či softvér pracuje správne, keď sa do existujúceho produktu implementuje nový modul alebo funkčnosť. Sanity testy sú dosť podobné smoke testom v tom, že umožňujú rýchle vyhodnotenie kvality vydania softvéru a určujú, či je spôsobilý pre ďalšie testovanie alebo nie. Zvyčajne sa však vykonávajú až po prijatí stabilného buildu softvéru alebo vtedy, keď sa v builde vykonali menšie zmeny v kóde alebo vo funkčnosti.

Regression testy – Tento typ testov pokrýva aj nezmenené časti softvéru s cieľom zistiť, či novo pridané funkcie ovplyvnili existujúcu funkcionalitu. Vykonávajú sa zväčša po signifikantných zmenách v aplikácii a kontrolujú sa pri nich všetky funkcie bez ohľadu na to, ktorá časť kódu sa zmenila. Pretože týchto testov býva najviac, zaberajú väčšinu času pri testovaní, a preto sú ideálnymi kandidátmi na automatizáciu.



Obr. 3.1: Diagram priebehu testovania

3.3 Unit, Integration a UI testovanie

Automatizáciu testov môžeme vykonávať rôznymi spôsobmi:

Unit testovanie – Jedným z týchto spôsobov je unit testovanie, ktoré testuje jednotlivé triedy a komponenty, avšak tento spôsob testovania môžu vykonávať iba vývojári, pretože vyžaduje znalosť zdrojového kódu.

Integration testovanie – Ďalším spôsobom je integration testovanie, ktoré testuje či komponenty pracujú správne ako celok, ale taktiež vyžaduje znalosť zdrojového kódu vďaka čomu ho môže vykonávať iba programátori.

UI testovanie – Poslednou variantou je UI testovanie, ktoré sa niekedy nazýva aj *„systémové testovanie“*. Tieto testy vykonávajú testeri, nevyžadujú znalosť zdrojového kódu (black box[2]) a ako už napovedá názov testujeme nimi chovanie celého systému a užívateľského rozhrania. Automatizáciu týchto testov som mal za úlohu vykonávať počas mojej praxe.

[2]systém so známymi vstupmi a výstupmi, ale s neznámou vnútornou štruktúrou a princípom fungovania

3.4 Exploratory testovanie

Projekt, na ktorom som pracoval bol dynamický a neustále sa menil a vyvíjal. Z toho dôvodu nebola ani presne definovaná softvérová špecifikácia, z ktorej by sa bežne dali dedukovať testovacie scenáre. V týchto prípadoch sa pri agilnej vývojovej metodike používa takzvaný *„exploratory testing“* (niekedy nazývaný aj *„ad-hoc*[3] *testing“*).

Pri tomto type testovania sa tester sám rozhoduje čo bude testovať a akým spôsobom, pričom si všetky testované scenáre stručne zapisuje spoločne s ich prioritou. Vďaka tomu je však kvalita testovania vo zvýšenej miere závislá od schopnosti testera vymýšľať testovacie prípady, nachádzať chyby a od jeho znalosti testovaného produktu. Tester sa musí často spoliehať na intuíciu a pýtať sa klienta otázky pre upresnenie produktovej špecifikácie.

Výsledkom exploratory testovania sú testovacie scenáre, ktoré je potom možné použiť pri ich automatizácii.

3.5 Automatizácia testov

Pred vydaním každej novej verzie aplikácie je dôležité, aby prešla aspoň základnou sadou testov, ktoré overujú jej fundamentálnu funkcionalitu, avšak ideálne by bolo ju overiť kompletne.

Vykonávanie týchto testov manuálnym spôsobom je časovo náročné a veľká časť z nich sa aj v priebehu vývoja opakuje. Ako som už spomenul v predošlej sekcii, tieto opakované testy voláme *„regression testy“*. Keďže počet týchto testov počas vývoja neustále narastá, po čase sa stane prakticky nemožné a neekonomické vykonávať ich manuálne. Na riešenie tohto problému sa preto používajú automatické testy.

Medzi hlavné výhody automatických testov patrí: [5]

Rýchlosť – automatické testy dokážu pokryť za rovnaký čas mnohonásobne viac scenárov ako manuálni testeri

Efektivita – keďže veľkú časť testov vykonajú automatické testy, manuálni testeri sa zatiaľ môžu venovať iným činnostiam

Precíznosť – automatické testy dokážu vykonávať testy opakovane rovnako, zatiaľ čo človek je nachylný ku chybám

Zníženie nákladov – vďaka automatickým testom je potrebných menej manuálnych testerov

Simulácia a emulácia – testovacie nástroje dokážu napodobňovať mnoho rôznych zariadení, ktoré by bežne využívali testovaný softvér

Neúprosnosť – automatické testy sa nikdy neunavia a môžu bežať prakticky neustále

[3] latinské slovné spojenie, ktoré znamená „na toto“ alebo „na tento účel“

Kapitola 4

Úlohy riešené počas bakalárskej praxe

Na moju pozíciu junior software engineer so zameraním na QA som nastúpil bez predchádzajúcich skúseností v tomto odvetví informatiky, takže moje prvé týždne boli zamerané hlavne na zoznámenie sa s projektom, na ktorom som pracoval, na osvojenie si používaných technológii, prieskum testovacích frameworkov, ktoré vyhovovali kritériám potrebným na realizáciu zadaných úloh a exploratory testovanie aplikácie.

4.1 Automatizačný framework

Pod pojmom framework si môžeme predstaviť nejakú zmes kódovacích štandardov, procesov, postupov, hierarchie projektov, mechanizmu predávania správ atď., ktoré je možné individuálne začleniť do vývoja alebo dodržiavať ako celok, čím sa využijú všetky výhody poskytované konkrétnym frameworkom.

Frameworky teda v krátkosti obsahujú funkcionality, ktoré sa zvyknú najčastejšie opakovať pri konkrétnom probléme. Vďaka tomu ich nie je potrebné viackrát implementovať a šetrí sa tak čas pri vývoji aplikácie, pretože programátor dorába už iba negenerickú funkcionalitu.

Vyskytujú sa však aj námietky, že používanie frameworku môže spomaliť beh aplikácie pretože nevieme ako efektívne sú tieto generické funkcionality naprogramované alebo že čas, ktorý by sme teoreticky ušetrili použitím cudzieho kódu, musíme aj tak venovať naštudovaniu frameworku. Často je však opak pravdou, pretože vďaka otvoreným zdrojovým kódom pracujú na týchto frameworkoch prakticky tisícky ľudí, v dôsledku čoho majú efektívny kód a dobrú dokumentáciu.

„Automatizačný framework“ je teda nejaký systém, ktorý bol vytvorený špeciálne na automatizáciu testov a poskytuje nám prostredie pre efektívny vývoj a vykonávanie našich testovacích skriptov. Zabezpečuje taktiež lepšiu škálovateľnosť, modularitu, zrozumiteľnosť, jednoduchšiu údržbu a podobne. Potreba jednotného testovacieho frameworku navyše vzrastá, keď viacero vývojárov pracuje na automatizovaní testov pre jednu aplikáciu a chceme sa vyhnúť situácii, kedy každý z vývojárov implementuje túto automatizáciu vlastným spôsobom.

Frameworky na automatizáciu webových aplikácii ktoré som používal počas mojej praxe fungujú na princípe simulovania reálneho užívateľského vstupu, a to tak, že zasielajú signály kliknutia myši a stlačenia kláves inštancii webového prehliadača, čím dokážu celkom dôveryhodne napodobniť skutočného užívateľa, ktorý by testovanú aplikáciu používal.

4.1.1 Výber testovacieho frameworku

Keďže mi bolo zadané, že na automatizovanie frontendových testov mám používať jazyk JavaScript, po preskúmaní rôznych možností som výber zredukoval na 3 frameworky ktorými boli Selenium, WebdriverIO a Cypress.

Selenium – Aj keď framework Selenium je asi najpopulárnejší a často poháňa aj iné automatizačné frameworky, je relatívne komplikované a zdĺhavé ho správne nastaviť. API ktoré používa je taktiež miestami zbytočne komplikované, čo by vo výsledku predstavovalo veľa času premrhaného písaním rôznych pomocných funkcii. Z týchto dôvodov pre mňa nebolo výhodné tento framework použiť.

Cypress – Framework Cypress je nováčik medzi testovacími frameworkmi, avšak aj za túto krátku dobu sa dokázal vďaka jeho jednoduchému nastaveniu, užívateľsky prívetivej dokumentácii a prehľadnému grafickému rozhraniu vyšplhať medzi top 3 JavaScriptové webové testovacie frameworky. Zásadnou nevýhodou tohto frameworku však je to, že podporuje iba prehliadače Chromium a Firefox, pričom podpora prehliadača Firefox bola pridaná iba nedávno. Toto bola prekážka, keďže v budúcnosti bolo plánované spúšťať automatické testy aj na operačnom systéme MacOS v prehliadači Safari. Ďalšia nevýhoda prichádzala v podobe platenej plnej verzie a niektorých chýbajúcich funkcionalít ako napríklad neschopnosť nahrávania súborov.

WebdriverIO – WebdriverIO je JavaScriptový automatizačný framework vytvorený na automatizáciu moderných webových a mobilných aplikácií využívajúci vlastnú implementáciu *„WebDriver“* protokolu, zásluhou čoho podporuje takmer všetky súčasné webové prehliadače. Zjednodušuje interakciu s testovanou aplikáciou vďaka jednoduchému API a poskytuje viaceré doplnky, vďaka ktorým je možné vytvoriť škálovateľné a robustné automatizované testy. Je taktiež jednoducho nastaviteľný vďaka zabudovanému „setup wizardu“. Aj keď dokumentácia jeho API nie je úplne dokončená, vybral som si práve tento framework, pretože s ním bolo možné automatizovať všetky scenáre, ktoré by teoreticky v aplikácii mohli nastať. Framework Webdriver.IO je taktiež aktívne vyvíjaný a kolegovia s ním už mali predchádzajúce praktické skúsenosti, vďaka čomu som mal v prípade problémov možnosť sa na nich obrátiť a požiadať ich o pomoc. [6]

4.1.2 Mocha

WebdriverIO nám umožňuje automaticky ovládať webový prehliadač, avšak samotné vykonávanie testov sprostredkuje framework *Mocha*, ktorý nám testy umožňuje rozdeliť do jednotlivých sád obsahujúcich viaceré scenáre a spúšťať ich nezávisle od seba. Dokáže taktiež spúšťať časti kódu pred a po vykonaní každého scenára alebo sady testov. [7]

```
describe('Login page tests', () => {
   before(() => {
      // Kód vykonaný pred celou sadou testov
   });

   beforeEach(() => {
      // Kód vykonaný pred každým scenárom
      login.open();

      expect('.header h1').dom.to.contain.text('Login');
   });

   it('Can log in using gmail account', () => {
      // Kód na prihlásenie pomocou gmail účtu
   });

   it('Can log in using facebook account', () => {
      // Kód na prihlásenie pomocou facebook účtu
   });
});
```

Výpis 4.1: Ukážka jednej sady testov vo frameworku Mocha

4.1.3 Chai

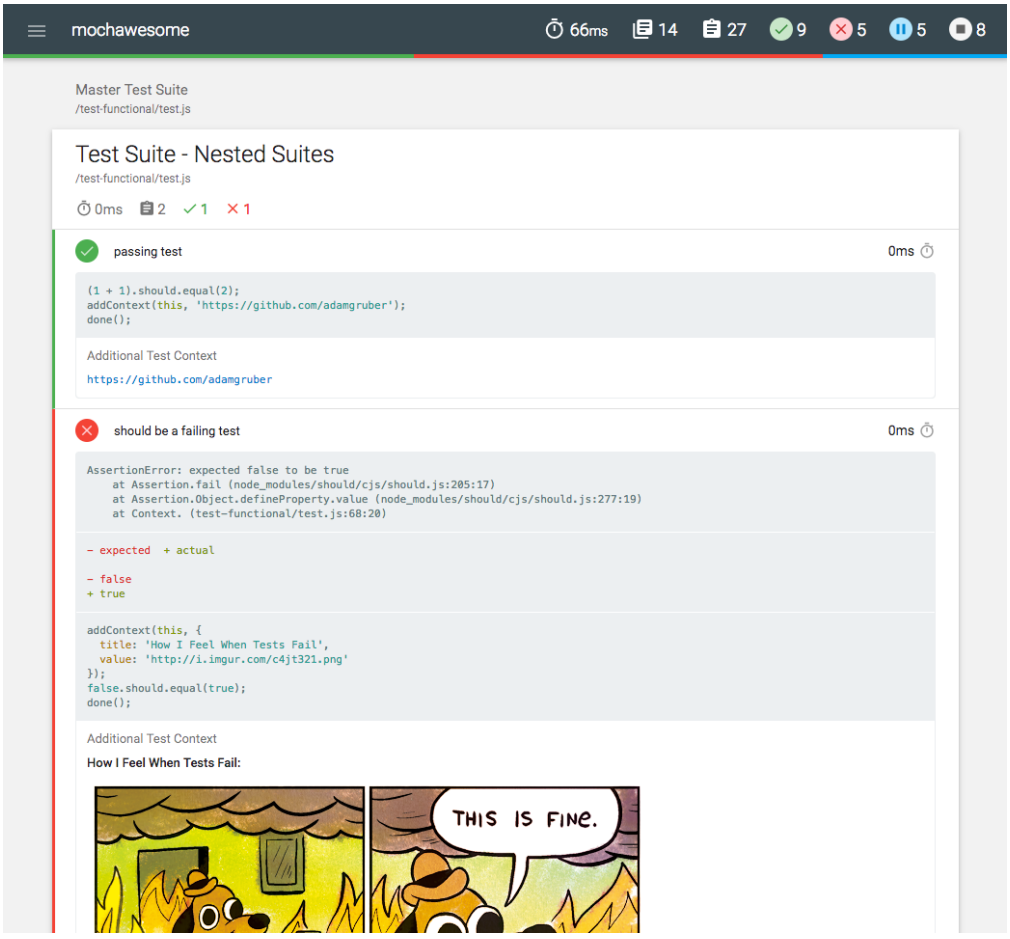
Pri automatických testoch sa využívajú aj takzvané *„assertions“*, čo sú prakticky podmienky, ktoré musia byť splnené, aby sa test vyhodnotil za prechodný. Knižnica *„Chai“* uľahčuje vytváranie týchto podmienok vďaka tomu, že poskytuje jednoduchú syntax, ktorá sa podobá písanému anglickému textu a dokáže aj interagovať s prvkami v Data Object Modeli HTML stránky. Túto syntax možeme vidieť aj v ukážke kódu 4.1.

4.1.4 Mochawesome

Po vykonaní testov si ich výsledok môžeme prezrieť vo výstupe z konzoly, avšak toto je nepraktické pre technicky nezdatných užívateľov a po nasadení na continuous integration je táto konzola aj ťažko prístupná. Klient, pre ktorého bola vyvíjaná aplikácia, na ktorej som pracoval chcel mať k výsledkom testov jednoduchý prístup, a preto som sa snažil nájsť riešenie na tento problém.

Nakoniec som sa rozhodol využiť knižnicu *„Mochawesome“*, ktorá z výsledkov testov vygeneruje jednoduchú a prehľadnú HTML stránku so všetkými potrebnými informáciami ako:

- počet prechádzajúcich testov
- počet zlyhaných testov
- čas vykonávania testov
- v prípade zlyhania testu aj výstup z konzoly a snímku obrazovky



Obr. 4.1: HTML stránka vygenerovaná reportérom Mochawesome

4.2 Frontend testy

Po nakonfigurovaní všetkých frameworkov a knižníc som začal implementovať konkrétne frontendové testy, teda testy užívateľského rozhrania. Na začiatok však bolo potrebné vytvoriť sadu nejakých testovacích scenárov a zoznámiť sa s jazykom JavaScript.

4.2.1 Tvorba testovacích scenárov

Ako som už spomínal, tvorba scenárov prebiehala pomocou exploratory testovania. Postupne som prechádzal aplikáciu a zapisoval som si tieto scenáre, ktoré som neskôr automatizoval.

V prípade, že som pri testovaní narazil na nejaký bug, nahlásil som ho do aplikácie *„Jira“*, ktorú sme pri práci využívali na zaznamenávanie a organizáciu chýb, návrhov a vylepšení. Táto aplikácia taktiež umožňuje integráciu s aplikáciou *Git*, vďaka čomu je možné ku chybe priradiť konkrétny commit, ktorý túto chybu spôsobil alebo opravil.

4.2.2 Jazyk JavaScript

Počas mojej praxe som na písanie testov vo frameworku WebdriverIO používal programovací jazyk JavaScript. JavaScript je ľahký, jednovláknový, interpretovaný alebo just-in-time kompilovaný programovací jazyk s prvotriednymi funkciami[1], ktorý podporuje objektovo-orientované, imperatívne a aj deklaratívne programovacie štýly. Má syntax a štruktúru, ktorá je podobná programovaciemu jazyku C. Aj keď je najznámejší ako skriptovací jazyk pre webové stránky, používa ho aj veľa prostredí, ktoré nie sú prehliadačmi, napríklad Node.js, Apache CouchDB a Adobe Acrobat. Práve Node.js, čo je medziplatformové, back-endové runtime prostredie JavaScriptu, som používal pri vývoji automatizačných testov. [8]

4.2.3 Dizajnový vzor Page-Object

Pri písaní testov som sa rozhodol využiť dizajnový vzor *„Page-Object“*, ktorý je na túto úlohu ideálny, pretože poskytuje spôsob izolovania skriptov jednotlivých testovacích scenárov od kódu rozhraní pomocou ktorých s testovanou stránkou interagujeme.

Cieľom tohto vzoru je abstrahovať akékoľvek informácie o stránke od konkrétnych testov. V ideálnom prípade by mali byť uložené všetky selektory a funkcie ktoré sú pre určitú stránku jedinečné v triede/objekte tejto stránky, aby sa kód testovacích scenárov nemusel prerábať aj po úplnom prepracovaní stránky. Ak je tento vzor správne realizovaný, bolo by po zmene dizajnu alebo iných úpravách potrebné iba a pozmeniť selektory alebo upraviť funkcie v objekte tejto stránky. Navyše sú vďaka tomuto vzoru výsledné testy stručnejšie a čitateľnejšie.

[1] prvotriedne funkcie sú funkcie, s ktorými sa zaobchádza ako s akoukoľvek inou premennou

```
class Page {
   open(path) {
      browser.url(path)
   }
}

class LoginPage extends Page {
   get usernameField() { return $('#username') }
   get passwordField() { return $('#password') }
   get submitButton() { return $('#login button[type=submit]') }

   open() {
      super.open('login')
   }

   submit() {
      this.submitButton.click()
   }
}
```

Výpis 4.2: Ukážka vzoru Page-Object

V ukážke tohto vzoru 4.2 je možné v triede *LoginPage* vidieť aj takzvané selektory. Vďaka týmto reťazcom je možné vyberať konkrétne prvky stránky a následne s nimi pracovať. Pri automatickom testovaní je najvýhodnejšie mať na každom objekte s ktorým pracujeme ID selektor, pretože sa môže nachádzať na stránke iba jeden a tým nám zaručí väčšiu stabilitu testov aj v prípade zmeny dizajnu stránky.

Pridelenie unikátneho ID však vyžaduje zásah do zdrojového kódu aplikácie a musia ho preto vykonávať vývojári. Pri tomto však nastali počas vytvárania testov niektoré problémy, keďže v aplikácii boli využívané aj predrobené grafické komponenty ktoré priraďovanie unikátneho ID nepodporovali, a preto som v niektorých prípadoch musel použiť selektory *XPath*, ktoré síce dokážu vybrať každý prvok na stránke ale prestanú fungovať aj po menších dizajnových zmenách, čo pri vyvíjajúcej sa aplikácii nie je ideálne.

```
#login-button
```

Výpis 4.3: ID selektor

```
//body/div[@id='root']/div[1]/ul[1]/a
```

Výpis 4.4: XPath selektor

4.2.4 Mockup dáta

Počas testovania rôznych funkcionalít importu dát som potreboval vytvoriť umelé dáta, ktoré nazývame aj *„Mockup dáta“*. Keďže som ich potreboval veľké množstvo, nebolo objektívne možné ich vytvoriť ručne.

Tieto dáta slúžili hlavne na import BEO a mali obsahovať mená, adresy, ceny služieb, emailové adresy, telefónne čísla a preto som hľadal knižnicu alebo službu ktorá by dokázala tieto kategórie dát vygenerovať automaticky a mohol som v nej špecifikovať aj formátovanie výsledného súboru.

Po preskúmaní viacerých možností som sa nakoniec rozhodol využiť službu ***mockaroo.com***, pomocou ktorej je možné vygenerovať veľké množstvo realisticky vyzerajúcich dát rôzneho typu. Zvolil som ju hlavne kvôli tomu, že oproti konkurenčným službám je možné dáta generovať aj zadarmo, na rozdiel od rôznych knižníc poskytuje jednoduché a prehľadné užívateľské rozhranie a taktiež disponuje webovým API pomocou ktorého sa dajú tieto dáta automaticky získavať podľa vopred špecifikovanej schémy.

4.2.5 Spúšťanie testov podľa dôležitosti

Pri väčšom počte automatických testov čas potrebný na ich vykonanie viditeľne stúpa. Hľadal som preto spôsob ako bežne spúšťať iba najdôležitejšie testy a celkové testovanie by prebehlo napríklad iba pred nasadením novej verzie aplikácie.

Ako som už spomenul, pri exploratory testovaní sa mimo popisu zapisovali aj priority testovaných scenárov. V našom prípade to bolo číslo od 1 do 5 pričom hodnota 1 až 3 boli testy s nižšou prioritou a zahŕňali regression testy a 4 až 5 boli naopak sanity testy s vyššou prioritou.

Pretože som nechcel testovacie scenáre rozdeľovať na viacero súborov, čo by skomplikovalo prácu s nimi a zhoršilo ich prehľadnosť, rozhodol som sa rozšíriť funkcionalitu frameworku *Mocha*. Keďže jazyk JavaScript umožňuje prepisovať už definované funkcie, modifikoval som funkciu, pomocou ktorej sa definujú jednotlivé scenáre, aby prijímala aj číslo od 1 do 5, ktoré určovalo jeho prioritu. Pri spúšťaní testov sa potom podľa hodnoty definovanej v premennej prostredia[2] spustili iba testy s vyššou prioritou ako táto hodnota.

4.2.6 Generovanie kostry kódu pre testy

Testovacie scenáre z exploratory testingu sme ukladali do aplikácie *Google Sheets*, avšak po dokončení práce na testovacej suite bolo potrebné tieto scenáre aj implementovať. Keďže táto činnosť zahrňovala aj veľa repetitívneho kopírovania textu z tabuliek a písanie boilerplate[3] kódu, zjednodušil som si ju vytvorením jednoduchého Pythonového skriptu, ktorý s pomocou šablónovacej knižnice *Jinja2* premenil CSV súbor s testovacími scenármi na kód testov podobný ako v ukážke 4.1.

[2] anglicky environment variable, je dynamická premenná poskytovaná operačným systémom viazaná na bežiaci proces

[3] kúsky kódu, ktoré sa opakujú na viacerých miestach s malými alebo žiadnymi obmenami

4.3 Continuous Integration a Continous Deployment

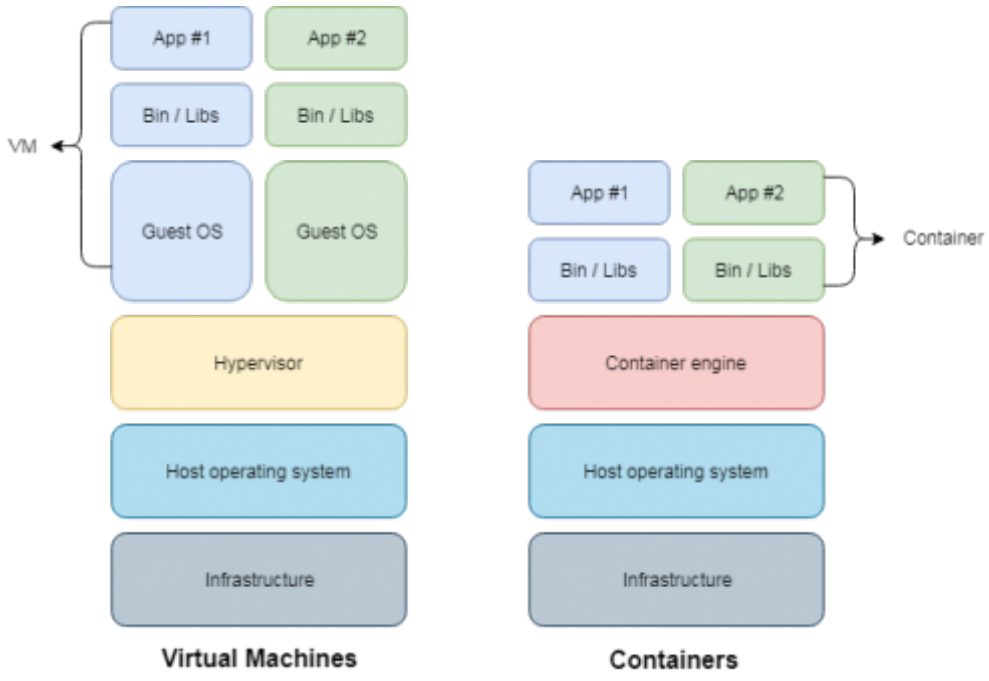
Pri vývoji aplikácie a testov sme používali distribuovaný systém riadenia revízií s názvom ***Git***, ktorý je aktuálne najpoužívanejší na svete. Kód môže byť uložený paralelne vo viacerých vetvách, pričom jedna je vždy označená ako hlavná. [9]

„Continuous Integration“ (v preklade *nepretržitá integrácia*), ďalej skratkou CI, funguje tak, že po každej prichádzajúcej zmene kódu na Git serveri sa spustí preddefinovaný skript, ktorý môže vykonávať rôzne činnosti ako napríklad kompilovať kód, spúšťať testy, overovať zmeny kódu a podobne.

„Continous Deployment“ (v preklade *nepretržité nasadzovanie*), ďalej skratkou CD, je len ďalší krok CI, ktorý automaticky skompiluje a nahradí stávajúcu verziu aplikácie novou po nahratí kódu na vetvu dedikovanú pre produkčný kód. [10]

4.3.1 Docker

Skoro žiaden program nefunguje bez operačného systému, rovnako aj pri CI a CD sú spomínané skripty spúšťané na operačnom systéme podľa nášho výberu s pomocou kontajnerizačného systému *„Docker“*.



Obr. 4.2: Porovnanie architektúry kontajnerov a virtuálnych strojov

Virtuálne stroje simulujú všetky časti počítača vrátane obrazovky a pevného disku, ktorý na našom skutočnom počítači (často označovanom ako hostiteľ) predstavuje iba veľký jeden súbor

(nazývaný virtuálny pevný disk). Na virtuálnom stroji so systémom Windows obsahuje virtuálny pevný disk všetok kód operačného systému Windows, čo môže byť niekoľko gigabajtov. Windows vo virtuálnom stroji nevie, že beží vo vnútri simulácie a nie skutočného počítača - iba si myslí, že je hlavný operačný systém.

Kontajnery v Dockeri sú dosť podobné virtuálnym strojom tým, že sú kompletne izolované od hostiteľského systému a majú virtualizovaný hardware. Spustenie niekoľkých inštancií toho istého operačného systému je nadbytočné a nepotrebné, preto na rozdiel od virtuálnych strojov tieto kontajnery zdieľajú nadbytočné zdroje, ako sú veľké súbory operačného systému a systémové jadro hostiteľa, vďaka čomu sú mnohonásobne menej náročné na výpočtovú silu a zaberajú oveľa menej miesta pri zachovaní rovnakej miery bezpečnosti. Medzi každým spusteným virtualizovaným programom majú kontajnery taktiež rozdelenú výpočtovú silu, čo znamená že sa nemôže stať že by jeden kontajner negatívne ovplyvňoval ostatné. [11]

Kontajnery sa vytvárajú pomocou súboru zvaného *Dockerfile*, v ktorom sú definované inštrukcie na zostrojenie kontajnerového obrazu. Pri vytváraní obrazu pre nasadenie do CI som sa rozhodol použiť Linuxovú distribúciu *„Alpine Linux“* z dôvodu jej malej veľkosti (iba okolo 100MB) a jednoduchosti (predinštalované sú iba esenciálne programy, všetko ostatné je potrebné doinštalovať pomocou súboru Dockerfile).

```
FROM node:lts-alpine

RUN apk add --update xvfb \
openjdk11-jre-headless \
chromium \
chromium-chromedriver \
findutils

RUN yarn global add selenium-standalone \
&& selenium-standalone install

COPY . .
```

Výpis 4.5: Ukážka Dockerfile súboru

Tento súbor som následne nahral na Docker Hub, kde sa skompiloval do kontajnerového obrazu, ktorý neskôr môžeme využiť v CI skripte.

4.3.2 GitLab

Na využívanie CI a CD je však potrebný Git server spolu s CI serverom, čo je však náročné na údržbu. Preto sme na tento účel využili platformu GitLab, ktorá obe tieto služby zahŕňa.

Vytvoril som jednoduchý CI skript, ktorý „roztočí“ spomenutý kontajnerový obraz, nainštaluje dependencie potrebné pre spúšťanie testov, vykoná testy a nakoniec nahrá HTML súbor s výsledkom testov vygenerovaných pomocou mochawesome ako artefakt na GitLab, kde je možné ho jednoducho preskúmať.

```
stages:
— test

Tests:
stage: test
image: <docker image name>
only:
— triggers
before_script:
— yarn install
script:
— yarn run test
after_script:
— find —name 'results_*' —type f —exec mv {} result.html \; —quit # Move the
    mochawesome result
artifacts:
when: always
expose_as: 'Mochawesome test results'
paths:
— result.html
```

Výpis 4.6: Ukážka GitLab CI skriptu

4.3.3 Manuálne spúšťanie CI testov

Po nasadení testov na CI však bolo potrebné, aby ich bolo možné občasne manuálne spúšťať (napríklad po nasadení nových testov alebo po zlyhaní nejakého testu). Túto funkcionalitu taktiež vyžadoval aj klient, s ktorým sme pracovali. V GitLab CI dokumentácii som našiel REST endpo-

int[4], pomocou ktorého toto bolo možné, a tak som vytvoril Bash skript ktorý tento endpoint zavolal a vypísal aj odkaz na výsledok testu.

```
#/bin/bash
if ! dpkg-query -W -f='${Status}' jq | grep "ok installed"; then sudo apt install
    jq; fi

web_url=$(curl -s -X POST -F token=<access token> -F ref=tests https://gitlab.com/
    api/v4/projects/<project id>/trigger/pipeline | jq --raw-output '.web_url')

if [ -z "$web_url" ] ; then
  echo "Failure"
  exit 1
else
  echo "CI Triggered: $web_url"
  xdg-open $web_url
fi
```

Výpis 4.7: Skript na manuálne spustenie GitLab CI testov

4.4 Backend testy

Keďže počas všetkých frontend testov sú na pozadí odosielané aj požiadavky na backend aplikácie, nebolo vyžadované jednotlivé endpointy testovať. Dostal som však za úlohu otestovať automatické škálovanie backendu hostovaného službou *Google Cloud*, čo znamenalo že som musel nájsť spôsob ako tento záťažový test vykonať.

Najprv som sa tento problém pokúsil vyriešiť spustením viacerých inštancií frontend testov paralelne, avšak roztočenie viacerých docker obrazov trvalo dlhú dobu a framework WebdriverIO nebol dostatočne rýchly na to, aby zahĺtil aplikáciu požiadavkami.

4.4.1 GraphQL

Testovaná aplikácia využívala na komunikáciu s backendom dátový dotazovací a manipulačný jazyk s názvom *GraphQL*. Endpointy využívajúce tento jazyk sa líšia od REST endpointov tak, že existuje iba jedno rozhranie na ktoré sa zasielajú dotazy, avšak *GraphQL* umožňuje špecifikovať polia, ktoré chceme vrátiť, vďaka čomu sú tieto požiadavky skoro vždy menšie a efektívnejšie. Tento jazyk taktiež umožňuje vývojárom načítať viacero entít v jednej požiadavke, čo ďalej zvyšuje efektivitu

[4]zjednodušene je to webová adresa reprezentujúca určitú funkciu, na ktorú zasielame požiadavky spolu s nejakými parametrami

každého dotazu. [12]

```
{
  human(id: "1000") {
    name
    height(unit: METER)
    age
  }
}
```

Výpis 4.8: Ukážka GraphQL dotazu

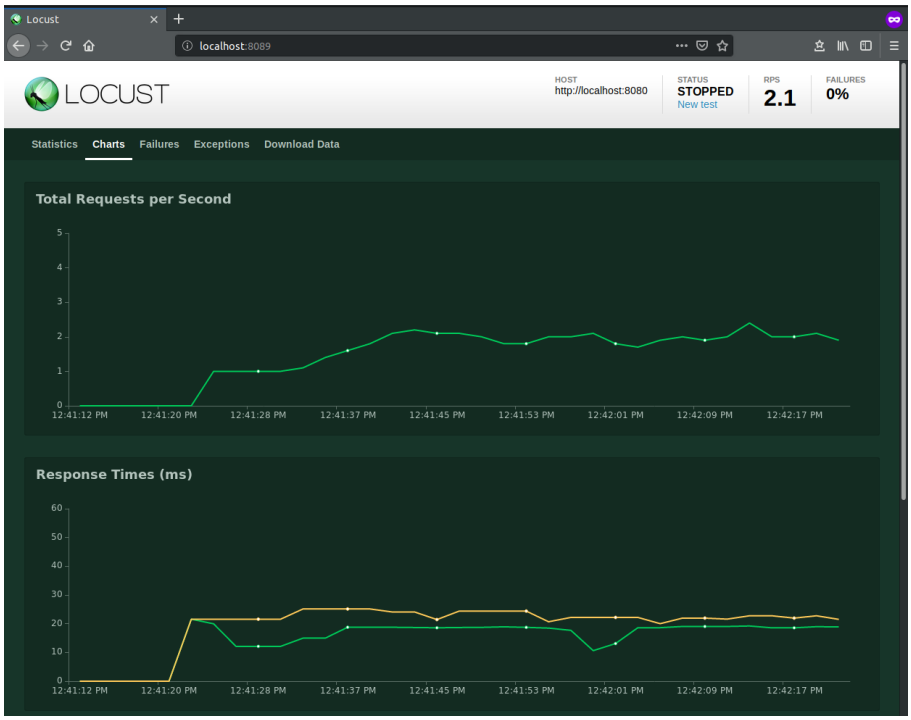
```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 1.72,
      "age": 33
    }
  }
}
```

Výpis 4.9: Ukážka GraphQL výsledku

Toto však znamenalo, že bolo teoreticky možné úplne obísť webovú aplikáciu a dotazy na zaťaženie backendu odosielať priamo.

4.4.2 Locust

Pythonový testovací nástroj Locust umožňuje špecifikovať a spúšťať záťažové scenáre pomocou Pythonového kódu a taktiež má prehľadné webové rozhranie, v ktorom je možné vidieť graf priebehu celého testovania.



Obr. 4.3: Ukážka webového rozhrania nástroja Locust

Napodobnil som teda v Python kóde volanie webovej aplikácie spolu s autentifikačným tokenom a zostrojil som zložitý *GraphQL* dotaz, ktorý čo najviac zaťažil backend aplikácie. Následne som vo webovom rozhraní Locustu spustil volanie tohto dotazu niekoľkokrát za sekundu z 500 paralelných inštancii.

```
from locust import HttpLocust, TaskSet, task

class GraphqlFlood(TaskSet):
    def on_start(self):
        self.flood()

    @task
    def flood(self):
        self.client.post(
            "http://api.app.com/graphql",
            name="GraphQL",
            headers={
                "Accept": "application/graphql",
                "Authorization": "<auth token>"
            },
            json={"query": "<graphql query>" }
        )

class LoadTest(HttpLocust):
    task_set = GraphqlFlood
```

Výpis 4.10: Ukážka Locust kódu na odosielanie GraphQL dotazov

Pomocou Locustu sa mi opakovaným odosielaním dotazov nakoniec podarilo zaťažiť backend, čím som nasimuloval podmienky pri zvýšenom počte užívateľov a podľa grafu priemernej odozvy backendu počas testovania som zistil, že automatické škálovanie fungovalo správne.

Kapitola 5

Záver

Za záver by som zhrnul skúsenosti z môjho bakalárskeho štúdia, ktoré som dokázal uplatniť, alebo ktoré mi naopak chýbali a taktiež opísal a zhodnotil moje dosiahnuté výsledky.

5.1 Teoretické a praktické znalosti a zručnosti získané v priebehu štúdia uplatnené študentom v priebehu odbornej praxe

Za najväčší prínos v rámci absolvovania bakalárskeho štúdia považujem nadobudnutie vedomostí v rámci predmetu *Tvorba aplikací pro mobilní zařízení 1*, kde som získal skúsenosti s jazykmi JavaScript a HTML, ktoré som potreboval pri práci s frameworkom *Webdriver.IO* počas písania frontendových automatických testov.

Pri písaní záťažových testov a rôznych drobných skriptov som využil znalosti z predmetu *Skriptovací programovací jazyky*, v ktorom som sa naučil pracovať v jazyku Python.

Počas komunikácie so zákazníkom a pri hľadaní poznatkov k novým technológiám som aplikoval moje znalosti technickej angličtiny z predmetov *Ab/I-FEI* až *Ab/IV-FEI*.

5.2 Znalosti či zručnosti chýbajúce študentovi v priebehu odbornej praxe

Nakoľko sa žiadny predmet v mojom bakalárskom študijnom programe nezaoberá konkrétne problematikou testovania softvéru, získal som značnú časť skúseností až počas praxe alebo vo voľnom čase.

V prvom rade by som podotkol hlavne technológiu *Git*, s ktorou sme sa síce stretli v predmete *Tvorba aplikací pro mobilní zařízení 2*, avšak nie v takom rozsahu ako som ju využíval pri vykonávaní mojej praxe. Následne mi chýbali znalosti ohľadom kontajnerizácie a nepretržitej integrácie, čo sú podľa môjho názoru v dnešnej dobe cloudových služieb dve veľmi používané technológie a bolo by užitočné ich ovládať.

5.3 Dosiahnuté výsledky v priebehu odbornej praxe a ich celkové zhodnotenie

Výsledkom mojej praxe bola sada testov, ktorú som následne aj nasadil do prostredia nepretržitej integrácie. Nepodarilo sa mi implementovať taký počet testovacích scenárov ako som pôvodne predpokladal, pretože sa špecifikácia aplikácie často menila. Zmena špecifikácie mala za následok spomalenie môjho progresu a často som musel modifikovať predtým fungujúci kód, ktorý po prepracovaní užívateľského rozhrania prestal fungovať. Aj napriek tomu si však myslím, že moja práca nebola zbytočná a pre firmu i zákazníka mala kladný prínos, pretože sa mi týmito testami podarilo zachytiť niekoľko chýb.

Vyskúšal som si prácu v tíme viacerých ľudí a naučil som sa kolektívne riešiť rôzne problémy a prekážky, ktoré sa vyskytli. Dozvedel som sa aké spôsoby testovania existujú a aký majú prínos pre koncový produkt. Navrhol som testovacie scenáre, ktoré som neskôr aj automatizoval, oboznámil som sa pri tom s mnohými novými technológiami a rozšíril som si svoje obzory v oblasti automatizácie testov a testovania softvéru.

Myslím si, že mi absolvovanie tejto praxe taktiež zvýšilo hodnotu na trhu práce, nakoľko som získal skúsenosti s celým cyklom agilného vývoja a prácou v reálnej firme.

Zoznam použitej literatúry

1. *Profiq* [online] [cit. 2021-03-30]. Dostupné z : `https://www.profiq.com/`.
2. *Student Pool* [online] [cit. 2021-03-30]. Dostupné z : `https://www.pracujprosiliconvalley.cz/student-pool/`.
3. *Planesty* [online] [cit. 2021-03-30]. Dostupné z : `https://www.planesty.com/`.
4. PRESSMAN, Roger S. *Software Engineering A Practitioner's Approach*. McGraw-Hill Education, 2014. ISBN 978-0078022128.
5. PATTON, Ron. *Software Testing*. Sams Publishing, 2006. ISBN 978-0672327988.
6. *What is Webdriver.IO?* [Online] [cit. 2021-03-30]. Dostupné z : `https://webdriver.io/docs/what-is-webdriverio`.
7. *Mocha - the fun, simple, flexible JavaScript test framework* [online] [cit. 2021-03-30]. Dostupné z : `https://mochajs.org/`.
8. *JavaScript* [online] [cit. 2020-02-03]. Dostupné z : `https://developer.mozilla.org/en-US/docs/Web/JavaScript`.
9. *Getting Started - What is Git?* [Online] [cit. 2021-03-30]. Dostupné z : `https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F`.
10. *GitLab CI/CD* [online] [cit. 2021-03-30]. Dostupné z : `https://docs.gitlab.com/ee/ci/README.html`.
11. *Docker a jeho použitie pri kontajnerizácii* [online] [cit. 2021-03-30]. Dostupné z : `https://magazin.kpi.fei.tuke.sk/2019/02/docker-a-jeho-pouzitie-pri-kontajnerizacii/`.
12. *GraphQL | A query language for your API* [online] [cit. 2021-03-30]. Dostupné z : `https://graphql.org/`.